CSC 347 - Concepts of Programming Languages

Functional Programming with Lists

Instructor: Stefan Mitsch

Learning Objectives

❓ How do we support processing collections?

- Understand functional collections processing
- Understand list comprehensions

Exercise: Print Every Element of a List

❓ Express in a functional style with pattern matching

```
1 def printList (xs:List[Int]) : Unit =
2
3
4
5
6
7
8 val xs = List(11,21,31)
9 printList (xs)
```

Exercise: Print Every Element of a List

❓ Visit all elements of a list

```
1 def printList (xs:List[Int]) : Unit = xs match
2   case Nil   => ()
3   case y::ys =>
4
5     printList (ys)
6
7
8 val xs = List(11,21,31)
9 printList (xs)
```

Exercise: Print Every Element of a List

❓ Print an element when visiting

```
1 def printList (xs:List[Int]) : Unit = xs match
2   case Nil   => ()
3   case y::ys =>
4     println (y)
5     printList (ys)
6
7
8 val xs = List(11,21,31)
9 printList (xs)
```

Exercise: Print Every Element of a List

❓ Format every element before printing

```
1 def printList (xs:List[Int]) : Unit = xs match
2   case Nil   => ()
3   case y::ys =>
4     println ("0x%02x".format(y))
5     printList (ys)
6
7
8 val xs = List(11,21,31)
9 printList (xs)
```

List Operation: Foreach

💡 Generalize the idea of processing every element

```
1 def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
2   case Nil    => ()
3   case y::ys =>
4     f (y)
5     foreach (ys, f)
6
7
8 val xs = List(11,21,31)
9 foreach (xs, println)
```

List Operation: Foreach

💡 Customize with changed function argument

```
1 def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
2   case Nil    => ()
3   case y::ys =>
4     f (y)
5     foreach (ys, f)
6
7 def printHex (x:Int) = println("0x%02x".format(x))
8 val xs = List(11,21,31)
9 foreach (xs, printHex)
```

List Operation: Foreach

💡 Generalize the type of list with parameters

```
1 def foreach [X] (xs:List[X], f:X=>Unit) : Unit = xs match
2   case Nil    => ()
3   case y::ys =>
4     f (y)
5     foreach (ys, f)
6
7 def printLength (xs:List[Int]) = println (xs.length)
8 val xss = List(List(11,21,31),List(),List(41,51))
9 foreach (xss, printLength)
```

List Operation: Foreach

💡 Use a lambda expression (anonymous function)

```
1 def foreach [X] (xs:List[X], f:X=>Unit) : Unit = xs match
2   case Nil    => ()
3   case y::ys =>
4     f (y)
5     foreach (ys, f)
6
7
8 val xss = List(List(11,21,31),List(),List(41,51))
9 foreach (xss, (xs:List[Int]) => println (xs.length))
```

List Operation: Foreach

Using the builtin `List` class `foreach` method

- Lambda expression:

```
1 xs.foreach ((x:Int) => println (x))
```

- Named method/function:

```
1 def print (x:Int) = println (x)
2 xs.foreach (print)
```

```
val print = (x:Int) => println (x)
xs.foreach (print)
```

- Types unnecessary if Scala can infer

```
1 xs.foreach (x => println ("0x%02x".format(x)))
```

Types and Function Parameters

```
1 def foreach [X] (xs:List[X], f:X=>Unit) : Unit = ...
```

- `X` is a *type parameter*
  - Type parameters in square brackets
  - Value parameters in round brackets
  - Types before values
- `f` is a parameter of *function type:* `(X=>Unit)`
  - takes an argument of type `X`
  - returns a result of type `Unit`

Anonymous Functions

- Element processing code is often a small snippet
- Using lambda notation

```
1 val add = (x:Int, y:Int) => x+y
2 add(11,21)
```

- Use underscore when parameters used exactly once

```
1 val add = (_:Int) + (_:Int)
2 add(11,21)
```

- Types may be inferred in some contexts

```
1 var add : (Int,Int)=>Int = _ + _
2 add(11,21)
```

Reference and Structural Equality

- Return a reference to a list

```
1 def reference [X] (xs:List[X]) : List[X] = xs
2
3
4
5 xs eq reference(xs) /* reference equality */
6 xs == reference(xs) /* value equality */
```

```
1 res1: Boolean = true
2 res2: Boolean = true
```

Reference and Structural Equality

- Return a copy of a list

```
1 def copy [X] (xs:List[X]) : List[X] = xs match
2   case Nil   => Nil
3   case y::ys => y::copy(ys)
4
5 xs eq copy(xs) /* reference equality */
6 xs == copy(xs) /* value equality */
```

```
1 res1: Boolean = false
2 res2: Boolean = true
```

List Operation: Map

❓ Return a transformed copy of a list

```
1 def map [X,Y] (xs:List[X], f:X=>Y) : List[Y] = xs match
2   case Nil    => Nil
3   case y::ys => f(y) :: map (ys)
4
5 val xs = List(11,21,31)
6 map(xs, (y:Int) => "0x%02x".format (y))
```

- every element in input results in an element in output
- 💡 `copy` is just a specialization of `map` with `f:X=>X = x=>x`
- 💡 `foreach` is just a specialization of `map` with `f:X=>Unit`
- Builtin `map`

```
1 xs.map ("0x%02x".format (_))
```

Examples

- Map a `List[List[Int]]` to a `List[Int]`: length of inner lists

```
1 List(List(11,21,31),List(),List(41,51)).map (_.length)
```

```
1 res1: List[Int] = List(3, 0, 2)
```

- Map a `List[String]` to a `List[Int]`: length of strings

```
1 List("hi", "it's", "me").map (_.length)
```

```
1 res1: List[Int] = List(2, 4, 2)
```

- Map a `List[Int]` to a `List[Int]`: increment each value

```
1 List(1, 2, 3, 4).map (_ + 1)
```

```
1 res1: List[Int] = List(2, 3, 4, 5)
```

List and Set Comprehensions

Set Comprehensions

$$\{(m,n)|m \in 0,\ldots,10 \land n \in 0,\ldots,10 \land m \leq n\}$$

List Comprehensions

- In many PLs
- SETL (1960s)
- Haskell

```
1 s = [ (m,n) | m <- [0..10], n <- [0..10], m <= n ]
```

- Scala

```
1 val s = for m <- 0 to 10; n <- 0 to 10; if m <= n yield (m,n)
```

- Python

```
1 {(m, n) for m in range (0,11) for n in range(0,11) if m <= n}
```

- JavaScript 1.7

For Comprehensions

Method `map`

```
1 xs.map (x => "0x%02x".format(x))
```

For Comprehension `yield`

```
1 for x <- xs yield "0x%02x".format(x)
```

Method `foreach`

```
1 xs.foreach (x => println ("0x%02x".format(x)))
```

For Comprehension `do`

```
1 for x <- xs do println ("0x%02x".format(x))
```

List Operation: Filter

❓ Copy only elements that satisfy a predicate `f`

```
1 def filter [X] (xs:List[X], f:X=>Boolean) : List[X] = xs match
2   case Nil            => Nil
3   case y::ys if f (y) => y :: filter (ys, f)
4   case _::ys          =>      filter (ys, f)
5
6 val zs = (0 to 7).toList
7 filter(zs, ((_:Int) % 3 != 0))
```

For Comprehensions

Method `filter`

```
1 zs.filter (z => z % 3 != 0) // shorter: zs.filter(_ % 3 != 0)
```

Nested Methods

```
1 zs.filter (z => z % 3 != 0).map (z => "0x%02x".format(z))
```

For Comprehension

```
1 for  z <- zs; if z % 3 != 0  yield z
```

Combined For Comprehension

```
1 for z <- zs; if z % 3 != 0 yield "0x%02x".format(z)
```

For Comprehensions

- Multiple iterators

```
1 val xss = List(List(11,21,31),List(),List(41,51))
2 for xs <- xss; x <- xs yield (x, xs.length)
```

```
1 res1: List[(Int, Int)] = List((11,3), (21,3), (31,3), (41,2), (51,2))
```

- Cross product of independent iterators

```
1 val xs = List(11,21,31)
2 val ys = List("a","b")
3 for x <- xs; y <- ys yield (x, y)
```

```
1 res1: List[(Int, String)] = List((11,a), (11,b), (21,a), (21,b), (31,a), (31,b))
```

```
1 (for x <- (1 to 7);
2       y <- (1 to 9) yield (x, y)).length
```

```
1 res1: Int = 63
```

For Comprehensions: Types

```
1 val xs = List(11,21,31)
2 val ys = List("a","b")
3 for x <- xs;
4     y <- ys yield (x, y)
```

- `xs : List[Int]`
- `ys : List[String]`
- Scala infers types for iterator variables

```
1 x : Int
2 y : String
```

- `yield` provides the type for the result

```
1 (x, y) : (Int, String)
2 for ... yield (x, y) : List[(Int, String)]
```

Summary

- Element-wise list processing is a universal, higher-order function
- Element-processing function is passed to `foreach`, `map`, etc. as argument
- Scala for-comprehensions are a loop-style way of expressing list comprehensions

List Operation: Flatten

❓ Revisit the copy operation

```
1 def copy    [X] (xs:List[List[X]]) : List[List[X]] = xs match
2   case Nil   => Nil
3   case y::ys => y ::  copy    (ys)
4
5 val xss = List(List(11,21,31),List(),List(41,51))
6 copy(xss)
```

```
1 res1: List(List(11,21,31),List(),List(41,51))
```

List Operation: Flatten

💡 Create a copy with a flat structure: replace `::` with `:::`

```
1 def flatten [X] (xs:List[List[X]]) : List[X] = xs match
2   case Nil    => Nil
3   case y::ys => y ::: flatten (ys)
4
5 val xss = List(List(11,21,31),List(),List(41,51))
6 flatten(xss)
```

```
1 res1: List(11,21,31,41,51)
```

List Operation: FlatMap

❓ Revisit method `map`

```
1 def map      [X,Y] (xs:List[X], f:X=>List[Y]) : List[List[Y]] = xs match
2   case Nil    => Nil
3   case y::ys => f(y) ::  map      (ys, f)
4
5 val as = List(3,0,2)
6 map    (as, (x:Int) => (1 to x).toList)
```

```
1 res1: List(List(1,2,3), List(), List(1,2))
```

List Operation: FlatMap

💡 Create a transformed list with a flat structure: replace `::` with `:::`

```
1 def flatMap [X,Y] (xs:List[X], f:X=>List[Y]) : List[Y] = xs match
2   case Nil    => Nil
3   case y::ys => f(y) ::: flatMap (ys, f)
4
5 val as = List(3,0,2)
6 flatMap(as, (x:Int) => (1 to x).toList)
```

```
1 res1: List(1,2,3,1,2):::Nil
```

List Operation: FlatMap

Argument and return types of `map` and `flatMap`

- `map: (List[X],X=>List[Y]) => List[List[Y]]`
  - Each element of result list is a `List[Y]`
  - Length of result = length of `xs`
- `flatMap: (List[X],X=>List[Y]) => List[Y]`
  - Each element of result list is a `Y`
  - Length of result = sum of lengths of each `List[Y]`

For Comprehensions

Method `flatMap`

```
1 xss.flatMap (x=>x) // same as xss.flatten
```

For Comprehension

```
1 for xs <- xss; x <- xs yield x
```